Spring整合Mybatis原理

前言

Spring整合MyBatis的原理也是一道非常高频的面试题， 下面我们一起来记录一下其中的原理， 主要是Spring是怎么管理MyBatis中的Mapper动态代理的。

Mybatis集成Spring:

1.Mybatis集成Spring的适配器源码下载:

1.https://github.com/mybatis/spring

下载时注意版本:

MyBatis-Spring	MyBatis	Spring Framework	Spring Batch	Java
2.0	3.5+	5.0+	4.0+	Java 8+
1.3	3.4+	3.2.2+	2.1+	Java 6+

2.为了在Spring源码中能够看到MyBatis的源码，需要将Mybatis的源码和MyBatis-Spring的源码 设置标识名称和添加安装源码到本地仓库的插件:

```
<artifactId>mybatis</artifactId>
<version>3.5.3-xsls</version>
<packaging>jar</packaging>
```

```
<artifactId>mybatis-spring</artifactId>
<version>2.0.3-xsls</version>
<packaging>jar</packaging>
```

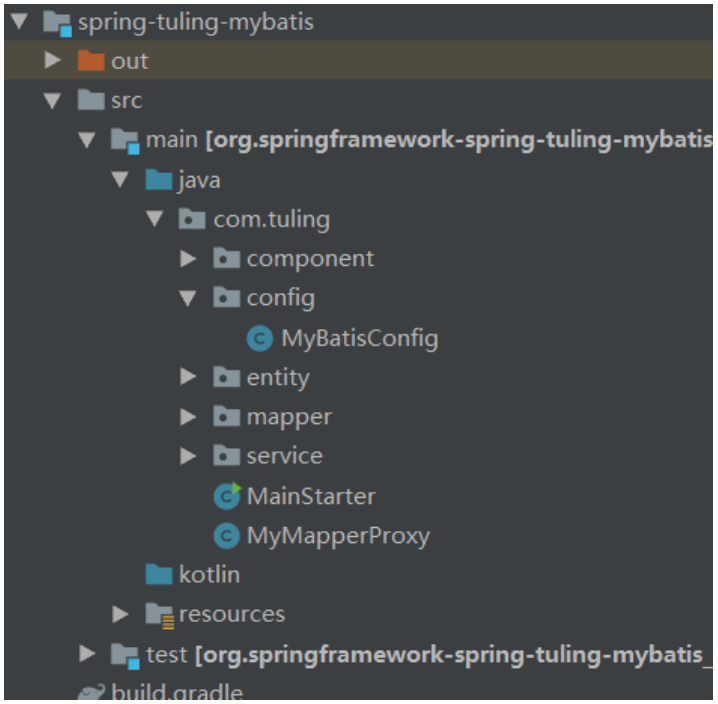
安装源码到本地仓库的插件:

```
<plugin>
  <artifactId>maven-source-plugin</artifactId>
  <version>3.0.1</version>
```

```
<configuration>
<attach>true</attach>
</configuration>
<executions>
<execution>
<phase>compile</phase>
 <goals>
 <goal>jar</goal>
 </goals>
 </execution>
 </executions>
</plugin>
```

2.Spring集成MyBatis

1.在Spring源码中添加Mybatis集成测试模块：



2.添加gradle依赖：

```

dependencies {
 testCompile group: 'junit', name: 'junit', version: '4.12'
 compile("mysql:mysql-connector-java:5.1.46")
 compile("com.alibaba:druid:1.1.8")
 compile("org.mybatis:mybatis-spring:2.0.3-xsls")
 compile("org.mybatis:mybatis:3.5.3-xsls")
 compile("org.projectlombok:lombok:1.18.4")
 compile("com.github.pagehelper:pagehelper:4.1.6")
  optional(project(":spring-context"))
  compile(project(":spring-jdbc"))
  compile("ch.qos.logback:logback-core:1.1.2")
  compile("ch.qos.logback:logback-classic:1.1.2")
```

```
    compile("org.slf4j:slf4j-api:1.7.7")
    optional(project(":spring-aop"))
    compile(project(":spring-jdbc"))
    compile("org.mybatis.caches:mybatis-ehcache:1.1.0")
    compile("net.sf.ehcache:ehcache-core:2.6.11")
}
```

3.添加mybatis的测试代码：

请参考提供的项目， 这里就不一一放代码了... 放上配置类：

```
@EnableTransactionManagement
@Configuration
@MapperScan(basePackages = {"com.tuling.mapper"})
@ComponentScan(basePackages = {"com.tuling"})
@Repository
public class MyBatisConfig { // =====> spring.xml


 /**
  * <bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource"> </bean>
  *
  * <bean class="org.mybatis.spring.SqlSessionFactoryBean" id="sqlSessionFactory">
  * datasource
  * mapper文件的路径
  * 别名
  *
  * </bean>
  *
  * <mapper-scan basePackage=""/>
  * @return
  * @throws IOException
  */
  @Bean // ===== > <bean class="org.mybatis.spring.SqlSessionFactoryBean">
  public SqlSessionFactoryBean sqlSessionFactory( ) throws IOException {
  SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
  factoryBean.setDataSource(dataSource());
  // 设置 MyBatis 配置文件路径
  factoryBean.setConfigLocation(new ClassPathResource("mybatis/mybatis-config.xml"));
  // 设置 SQL 映射文件路径
  factoryBean.setMapperLocations(new
PathMatchingResourcePatternResolver().getResources("classpath:mybatis/mapper/*.xml"));
  factoryBean.setTypeAliases(User.class);

  return factoryBean;


  }

```

```
38  public DataSource dataSource() {
39  DruidDataSource dataSource = new DruidDataSource();
40  dataSource.setUsername("root");
41  dataSource.setPassword("123456");
42  dataSource.setDriverClassName("com.mysql.jdbc.Driver");
43  dataSource.setUrl("jdbc:mysql://localhost:3306/mybatis_example");
44  return dataSource;
45  }
```

1.SqlSessionFactoryBean

```
1 public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>, Initializin
gBean, ApplicationListener<ApplicationEvent> {
```

实现FactoryBean接口的getObject方法：

```
1   /**
2   *
3   * 将SqlSessionFactory对象注入spring容器
4   * {@inheritDoc}
5   */
6   @Override
7   public SqlSessionFactory getObject() throws Exception {
8   if (this.sqlSessionFactory == null) {
9   afterPropertiesSet();
10   }
11
12   return this.sqlSessionFactory;
13   }
```

SqlSessionFactoryBean实现InitializingBean接口，需要实现其afterPropertiesSet()：

```
1   /**
2   * {@inheritDoc}
3   */
4 /**
5   * 方法实现说明:我们自己配置文件中配置了SqlSessionFactoryBean,我们发现配置了 该类实现了Factory
ean接口，也实现了bean的生命周期回调接口InitializingBean
6   * 首先我们会调用生命周期的回调afterPropertiesSet() 就是我们的SqlSessionFactorybean已经调用了
构造方法，已经调用了 我们的
7   *
8   * @author:xsls
9   * @return:
10   * @exception:
11   * @date:2019/8/23 19:33
12   */
13 @Override
14 public void afterPropertiesSet() throws Exception {
15   notNull(dataSource, "Property 'dataSource' is required");
16   notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");
17   state((configuration == null && configLocation == null) || !(configuration != null && c
onfigLocation != null),
```

```
  "Property 'configuration' and 'configLocation' can not specified with together");

  /**
  * 通过sqlSessionFactoryBuilder来构建我们的sqlSessionFactory
  */
  this.sqlSessionFactory = buildSqlSessionFactory();
 }

```

核心是buildSqlSessionFactory：

```

/**
 * 方法实现说明:构建我们的sqlSessionFactory的实例
 *
 * @author:xsls
 * @return:
 * @exception:
 * @date:2019/8/23 20:06
 */
protected SqlSessionFactory buildSqlSessionFactory() throws Exception {

 // 声明一个Configuration对象用于保存mybatis的所有的配置信息
 final Configuration targetConfiguration;

 XMLConfigBuilder xmlConfigBuilder = null;
 // 初始化 configuration 对象，和设置其 `configuration.variables` 属性
 /**
 * 判断当前的SqlSessionFactoryBean是否在配置@Bean的时候 factoryBean.setConfiguration();
 *
 */
 if (this.configuration != null) {
 /**
 * 把配置的SqlSessionFactoryBean配置的configuration 赋值给targetConfiguration
 */
 targetConfiguration = this.configuration;
 if (targetConfiguration.getVariables() == null) {
 targetConfiguration.setVariables(this.configurationProperties);
 } else if (this.configurationProperties != null) {
 targetConfiguration.getVariables().putAll(this.configurationProperties);
 }
 }
 /**
 * 对configLocation进行非空判断，由于我们配置了SqlSessionFactoryBean的configLocation属性设置
 *
 * @Bean public SqlSessionFactoryBean sqlSessionFactory( ) throws IOException { SqlSessi
onFactoryBean factoryBean
 * =new SqlSessionFactoryBean(); factoryBean.setDataSource(dataSource()); factoryBean.se
tConfigLocation(new
```

```
37   * ClassPathResource("mybatis/mybatis-config.xml")); factoryBean.setMapperLocations(new
38   *
PathMatchingResourcePatternResolver().getResources("classpath:mybatis/mapper/*.xml")); retu
rn factoryBean;
39   * }
40   */
41
42   else if (this.configLocation != null) {
43   /**
44   * 创建我们xml配置构建器对象,对mybatis/mybatis-config.xml配置文件进行解析 在这里以及把我们的my
baits-config.xml解析出要给document对象
45   */
46   xmlConfigBuilder = new XMLConfigBuilder(this.configLocation.getInputStream(), null, thi
s.configurationProperties);
47   /**
48   * 因为我们在创建XMLConfigBuilder的时候已经把我们的Configuration对象创建出来了
49   */
50   targetConfiguration = xmlConfigBuilder.getConfiguration();
51   } else {
52   LOGGER.debug(
53   () -> "Property 'configuration' or 'configLocation' not specified, using default MyBati
s Configuration");
54   targetConfiguration = new Configuration();
55   /**
56   * 判断configurationProperties不为空,那么就调用targetConfiguration.set方法 把configurationP
roperties注入到Configuration对象中
57   */
58   Optional.ofNullable(this.configurationProperties).ifPresent(targetConfiguration::setVar
iables);
59   }
60
61   /**
62   * objectFactory不为空,那么就调用targetConfiguration.set方法 把objectFactory注入到Configura
tion对象中
63   */
64
Optional.ofNullable(this.objectFactory).ifPresent(targetConfiguration::setObjectFactory);
65   /**
66   * objectWrapperFactory不为空,那么就调用targetConfiguration.set方法把 ObjectWrapperFactory
注入到Configuration对象中
67   */
68   Optional.ofNullable(this.objectWrapperFactory).ifPresent(targetConfiguration::setObject
WrapperFactory);
69
70   /**
71   * vfs不为空,那么就调用targetConfiguration.set方法把 vfs注入到Configuration对象中
72   */
73   Optional.ofNullable(this.vfs).ifPresent(targetConfiguration::setVfsImpl);
74
75   /**
```

```
76   * typeAliasesPackage配置情况分为二种 1) 在mybaits-config.xml中配置了(mybatis的方式) <typeA
liases>
77   * <package name="com.tuling.entity"></package> </typeAliases>
78   * 2)在配置我们的SqlSessionFactoryBean的时候配置了(Spring整合mybatis的方式)
79   *
80   * @Bean public SqlSessionFactoryBean sqlSessionFactory( ) throws IOException { SqlSessi
onFactoryBean factoryBean =
81   * new SqlSessionFactoryBean(); factoryBean.setDataSource(dataSource()); // 设置 MyBatis
配置文件路径
82   * factoryBean.setConfigLocation(new ClassPathResource("mybatis/mybatis-config.xml"));
// 设置 SQL 映射文件路径
83   * factoryBean.setMapperLocations(new
84   *
PathMatchingResourcePatternResolver().getResources("classpath:mybatis/mapper/*.xml"));
85   *
86   * factoryBean.setTypeAliasesPackage("com.tuling.entity"); return factoryBean; }
87   *
88   *
89   * 那么在Dept 就不需要写成com.tuling.entity了 <select id="findOne" parameterType="Integer"
resultType="Dept"> select *
90   * from dept where id = #{id} </select>
91   *
92   * 若我们在配置SqlSessionFactoryBean接口的时候配置了typeAliasesPackage 那么
93   * 这里才不会为空,同理,我们可以通过SqlSessionFactoryBean的typeAliasesSuperType 来控制哪些类的
别名不支持
94   */
95   if (hasLength(this.typeAliasesPackage)) {
96
97   /**
98   * 第一步:扫描我们typeAliasesPackage 包路径下的所有的实体类的class类型 第二步:进行过滤,然后注
册到Configuration的别名映射器中
99   */
100   scanClasses(this.typeAliasesPackage, this.typeAliasesSuperType).stream()
101   .filter(clazz -> !clazz.isAnonymousClass()).filter(clazz -> !clazz.isInterface())
102   .filter(clazz -> !clazz.isMemberClass()).forEach(targetConfiguration.getTypeAliasRegis
ry()::registerAlias);
103   }
104
105   /**
106   * 判断我们SqlSessionFactory是否配置了typeAliases(class类型) 一般typeAliasesPackage配置好了
就没有必要配置typeAliases
107   * 注册到Configuration的别名映射器中
108   */
109   if (!isEmpty(this.typeAliases)) {
110   Stream.of(this.typeAliases).forEach(typeAlias -> {
111   targetConfiguration.getTypeAliasRegistry().registerAlias(typeAlias);
112   LOGGER.debug(() -> "Registered type alias: '" + typeAlias + "'");
113   });
114   }
115
```

```
116   /**
117   * 把我们自定义的插件注册到我们的mybatis的配置类上 系统默认的插件 Executor (update, query, f
ushStatements, commit, rollback, getTransaction,
118   * close, isClosed) ParameterHandler (getParameterObject, setParameters) ResultSetHandl
r (handleResultSets,
119   * handleOutputParameters) StatementHandler (prepare, parameterize, batch, update,
query)
120   */
121   if (!isEmpty(this.plugins)) {
122   Stream.of(this.plugins).forEach(plugin -> {
123   targetConfiguration.addInterceptor(plugin);
124   LOGGER.debug(() -> "Registered plugin: '" + plugin + "'");
125   });
126   }
127
128   /**
129   * 扫描我们自定义的类型处理器(用来处理我们的java类型和数据库类型的转化) 并且注册到我们的 targe
Configuration(批量注册)
130   */
131   if (hasLength(this.typeHandlersPackage)) {
132   scanClasses(this.typeHandlersPackage, TypeHandler.class).stream().filter(clazz -> !cla
z.isAnonymousClass())
133   .filter(clazz -> !clazz.isInterface()).filter(clazz -> !Modifier.isAbstract(clazz.getM
difiers()))
134   .forEach(targetConfiguration.getTypeHandlerRegistry()::register);
135   }
136
137   /**
138   * 通过配置<TypeHandlers></TypeHandlers>的形式来注册我们的类型处理器对象
139   */
140   if (!isEmpty(this.typeHandlers)) {
141   Stream.of(this.typeHandlers).forEach(typeHandler -> {
142   targetConfiguration.getTypeHandlerRegistry().register(typeHandler);
143   LOGGER.debug(() -> "Registered type handler: '" + typeHandler + "'");
144   });
145   }
146
147   /**
148   * MyBatis 从 3.2 开始支持可插拔的脚本语言， 因此你可以在插入一种语言的驱动（language drive
r）之后来写基于这种语言的动态 SQL 查询
149   * 具体用法:博客地址:https://www.jianshu.com/p/5c368c621b89
150   */
151   if (!isEmpty(this.scriptingLanguageDrivers)) {
152   Stream.of(this.scriptingLanguageDrivers).forEach(languageDriver -> {
153   targetConfiguration.getLanguageRegistry().register(languageDriver);
154   LOGGER.debug(() -> "Registered scripting language driver: '" + languageDriver + "'");
155   });
156   }
157   Optional.ofNullable(this.defaultScriptingLanguageDriver)
```

```
158   .ifPresent(targetConfiguration::setDefaultScriptingLanguage);
159
160   /**
161   * 设置数据库厂商
162   */
163   if (this.databaseIdProvider != null) {// fix #64 set databaseId before parse mapper xml
s
164   try {
165
targetConfiguration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dataSource));
166   } catch (SQLException e) {
167   throw new NestedIOException("Failed getting a databaseId", e);
168   }
169   }
170
171   /**
172   * 若二级缓存不为空,注册二级缓存
173   */
174   Optional.ofNullable(this.cache).ifPresent(targetConfiguration::addCache);
175
176   if (xmlConfigBuilder != null) {
177   try {
178   /**
179   * 真正的解析我们的配置(mybatis-config.xml)的document对象
180   */
181   xmlConfigBuilder.parse();
182   LOGGER.debug(() -> "Parsed configuration file: '" + this.configLocation + "'");
183   } catch (Exception ex) {
184   throw new NestedIOException("Failed to parse config resource: " + this.configLocation,
ex);
185   } finally {
186   ErrorContext.instance().reset();
187   }
188   }
189
190   /**
191   * 为我们的configuration设置一个环境变量
192   */
193   targetConfiguration.setEnvironment(new Environment(this.environment,
194   this.transactionFactory == null ? new SpringManagedTransactionFactory() : this.transac
ionFactory,
195   this.dataSource));
196
197   /**
198   * 循环我们的mapper.xml文件
199   */
200   if (this.mapperLocations != null) {
201   if (this.mapperLocations.length == 0) {
```

```
202   LOGGER.warn(() -> "Property 'mapperLocations' was specified but matching resources are
not found.");
203   } else {
204   for (Resource mapperLocation : this.mapperLocations) {
205   if (mapperLocation == null) {
206   continue;
207   }
208   try {
209   /**
210   * 真正的循环我们的mapper.xml文件
211   */
212   XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.getInputStream(),
213   targetConfiguration, mapperLocation.toString(), targetConfiguration.getSqlFragments());
214   xmlMapperBuilder.parse();
215   } catch (Exception e) {
216   throw new NestedIOException("Failed to parse mapping resource: '" + mapperLocation +
"'", e);
217   } finally {
218   ErrorContext.instance().reset();
219   }
220   LOGGER.debug(() -> "Parsed mapper file: '" + mapperLocation + "'");
221   }
222   }
223   } else {
224   LOGGER.debug(() -> "Property 'mapperLocations' was not specified.");
225   }
226
227   /**
228   * 通过建造者模式构建我们的SqlSessionFactory对象 默认是DefaultSqlSessionFactory
229   */
230   return this.sqlSessionFactoryBuilder.build(targetConfiguration);
231 }
```

可知SqlSessionFactoryBean主要通过对applicationContext.xml解析完成时Configuration的实例化以及对完成对映射配置文件mapper*.xml的解析。

关键点：

1. *XMLConfigBuilder：在mybatis中主要负责解释mybatis-config.xml
    a. 解析完后，如果我们自己设置了则使用我们的设置的进行覆盖，不做一一介绍了
2. XMLMapperBuilder：负责解析映射配置文件
3. targetConfiguration.setEnvironment 这里注意一下 ， 事务工厂会使用一个新的new SpringManagedTransactionFactory()

而不是MyBatis之前的ManagedTransactionFactory. 这个SpringManagedTransactionFactory会使用Spring事务中的dataSource， 从而达到跟事务集成

2.Spring是怎么管理Mapper接口的动态代理的

https://www.processon.com/view/link/5f153429e401fd2e0deefd01

Spring和Mybatis时，我们重点要关注的就是这个代理对象。因为整合的目的就是：**把某个Mapper的代理对象作为一个bean放入Spring容器中，使得能够像使用一个普通bean一样去使用这个代理对象，比如能被@Autowire自动注入。**

比如当Spring和Mybatis整合之后，我们就可以使用如下的代码来使用Mybatis中的代理对象了：

```
@Component
public class UserService {
 @Autowired
 private UserMapper userMapper;

 public User getUserById(Integer id) {
 return userMapper.selectById(id);
 }
}
```

UserService中的userMapper属性就会被自动注入为Mybatis中的代理对象。如果你基于一个已经完成整合的项目去调试即可发现，userMapper的类型为：org.apache.ibatis.binding.MapperProxy@41a0aa7d。证明确实是Mybatis中的代理对象。

好，那么现在我们要解决的问题的就是：**如何能够把Mybatis的代理对象作为一个bean放入Spring容器中?**

要解决这个，我们需要对Spring的bean生成过程有一个了解。

Spring中Bean的产生过程

Spring启动过程中，大致会经过如下步骤去生成bean

1. 扫描指定的包路径下的class文件
2. 根据class信息生成对应的BeanDefinition
3. 在此处，程序员可以利用某些机制去修改BeanDefinition
4. 根据BeanDefinition生成bean实例
5. 把生成的bean实例放入Spring容器中

假设有一个A类，假设有如下代码：

一个A类：

```
@Component
public class A {
}
```

一个B类，不存在@Component注解

```
public class B {
}
```

执行如下代码：

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppCo
nfig.class);
System.out.println(context.getBean("a"));
```

输出结果为：com.tulingxueyuan.beans.A@6acdbdf5

A类对应的bean对象类型仍然为A类。但是这个结论是不确定的，我们可以利用BeanFactory后置处理器来修改BeanDefinition，我们添加一个BeanFactory后置处理器：

```
@Component
```

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
 @Override
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws B
eansException {
 BeanDefinition beanDefinition = beanFactory.getBeanDefinition("a");
 beanDefinition.setBeanClassName(B.class.getName());
 }
}
```

这样就会导致，原本的A类对应的BeanDefiniton被修改了，被修改成了B类，那么后续正常生成的bean对象的类型就是B类。此时，调用如下代码会报错：

```
context.getBean(A.class);
```

但是调用如下代码不会报错，尽管B类上没有@Component注解：

```
context.getBean(B.class);
```

并且，下面代码返回的结果是：com.tulingxueyuan.beans.B@4b1c1ea0

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppCo
nfig.class);
System.out.println(context.getBean("a"));
```

之所以讲这个问题，是想说明一个问题：**在Spring中，bean对象跟class没有直接关系，跟BeanDefinition才有直接关系。**

那么回到我们要解决的问题：**如何能够把Mybatis的代理对象作为一个bean放入Spring容器中?**

在Spring中，**如果你想生成一个bean，那么得先生成一个BeanDefinition**，就像你想new一个对象实例，得先有一个class。

解决问题

继续回到我们的问题，我们现在想自己生成一个bean，那么得先生成一个BeanDefinition，只要有了BeanDefinition，通过在BeanDefinition中设置**bean对象的类型**，然后把BeanDefinition添加给Spring，Spring就会根据BeanDefinition自动帮我们生成一个类型对应的bean对象。

所以，现在我们要解决两个问题：

1. **Mybatis的代理对象的类型是什么？因为我们要设置给BeanDefinition**
2. **我们怎么把BeanDefinition添加给Spring容器?**

注意：上文中我们使用的BeanFactory后置处理器，他只能修改BeanDefinition，并不能新增一个BeanDefinition。我们应该使用Import技术来添加一个BeanDefinition。后文再详细介绍如果使用Import技术来添加一个BeanDefinition，可以先看一下伪代码实现思路。

假设：我们有一个UserMapper接口，他的代理对象的类型为UserMapperProxy。
那么我们的思路就是这样的，伪代码如下：

```
BeanDefinitoin bd = new BeanDefinitoin();
bd.setBeanClassName(UserMapperProxy.class.getName());
SpringContainer.addBd(bd);
```

但是，这里有一个严重的问题，就是上文中的UserMapperProxy是我们假设的，他表示一个代理类的类型，然而Mybatis中的代理对象是利用的JDK的动态代理技术实现的，也就是代理对象的代理类是动态生成的，我们根本无法确定代理对象的代理类到底是什么。

所以回到我们的问题：**Mybatis的代理对象的类型是什么?**

本来可以有两个答案：

1. 代理对象对应的代理类
2. 代理对象对应的接口

那么答案1就相当于没有了，因为是代理类是动态生成的，那么我们来看答案2：**代理对象对应的接口**

如果我们采用答案2，那么我们的思路就是：

```
BeanDefinition bd = new BeanDefinitoin();
// 注意这里，设置的是UserMapper
bd.setBeanClassName(UserMapper.class.getName());
SpringContainer.addBd(bd);
```

但是，实际上给BeanDefinition对应的类型设置为一个接口是**行不通**的，因为Spring没有办法根据这个BeanDefinition去new出对应类型的实例，接口是没法直接new出实例的。

那么现在问题来了，我要解决的问题：**Mybatis的代理对象的类型是什么?**
两个答案都被我们否定了，所以这个问题是无解的，所以我们不能再沿着这个思路去思考了，只能回到最开始的问题：**如何能够把Mybatis的代理对象作为一个bean放入Spring容器中?**

总结上面的推理：**我们想通过设置BeanDefinition的class类型，然后由Spring自动的帮助我们去生成对应的bean，但是这条路是行不通的。**

终极解决方案

那么我们还有没有其他办法，可以去生成bean呢？并且**生成bean的逻辑不能由Spring来帮我们做**了，得由我们自己来做。

FactoryBean

有，那就是Spring中的FactoryBean。我们可以利用FactoryBean去自定义我们要生成的bean对象，比如：

```
@Component
public class MyFactoryBean implements FactoryBean {
 @Override
 public Object getObject() throws Exception {
 Object proxyInstance = Proxy.newProxyInstance(MyFactoryBean.class.getClassLoader(), new Class[]{UserMapper.class}, new InvocationHandler() {
 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 if (Object.class.equals(method.getDeclaringClass())) {
 return method.invoke(this, args);
 } else {
 // 执行代理逻辑
 return null;
 }
 }
 });

 return proxyInstance;
 }

 @Override
 public Class<?> getObjectType() {
```

```
 return UserMapper.class;
 }
}
```

我们定义了一个MyFactoryBean，它实现了FactoryBean，getObject方法就是用来自定义生成bean对象逻辑的。

执行如下代码：

```
public class Test {
 public static void main(String[] args) {
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppC
onfig.class);
 System.out.println("myFactoryBean: " + context.getBean("myFactoryBean"));
 System.out.println("&myFactoryBean: " + context.getBean("&myFactoryBean"));
 System.out.println("myFactoryBean-class: " +
context.getBean("myFactoryBean").getClass());
 }
}
```

将打印：

myFactoryBean: com.tulingxueyuan.beans.myFactoryBean$1@4d41cee

&myFactoryBean: com.tulingxueyuan.beans.myFactoryBean@3712b94

myFactoryBean-class: class com.sun.proxy.$Proxy20

从结果我们可以看到，从Spring容器中拿名字为"myFactoryBean"的bean对象，就是我们所自定义的jdk动态代理所生成的代理对象。

所以，我们可以通过FactoryBean来向Spring容器中添加一个自定义的bean对象。上文中所定义的MyFactoryBean对应的就是UserMapper，表示我们定义了一个MyFactoryBean，相当于把UserMapper对应的代理对象作为一个bean放入到了容器中。

但是作为程序员，我们不可能每定义了一个Mapper，还得去定义一个MyFactoryBean，这是很麻烦的事情，我们改造一下MyFactoryBean，让他变得更通用，比如：

```
@Component
public class MyFactoryBean implements FactoryBean {

 // 注意这里
 private Class mapperInterface;
 public MyFactoryBean(Class mapperInterface) {
 this.mapperInterface = mapperInterface;
 }

 @Override
 public Object getObject() throws Exception {
 Object proxyInstance = Proxy.newProxyInstance(MyFactoryBean.class.getClassLoader(), new
Class[]{mapperInterface}, new InvocationHandler() {
 @Override
```

```
14  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
15
16  if (Object.class.equals(method.getDeclaringClass())) {
17  return method.invoke(this, args);
18  } else {
19  // 执行代理逻辑
20  return null;
21  }
22  }
23  });
24
25  return proxyInstance;
26  }
27
28  @Override
29  public Class<?> getObjectType() {
30  return mapperInterface;
31  }
32 }
```

改造MyFactoryBean之后，MyFactoryBean变得灵活了，可以在构造MyFactoryBean时，通过构造传入不同的Mapper接口。

实际上MyFactoryBean也是一个Bean，我们也可以通过生成一个BeanDefinition来生成一个MyFactoryBean，并给构造方法的参数设置不同的值，比如伪代码如下：

```
1 BeanDefinition bd = new BeanDefinitoin();
2 // 注意一：设置的是MyFactoryBean
3 bd.setBeanClassName(MyFactoryBean.class.getName());
4 // 注意二：表示当前BeanDefinition在生成bean对象时，会通过调用MyFactoryBean的构造方法来生成，并传入UserMapper
5 bd.getConstructorArgumentValues().addGenericArgumentValue(UserMapper.class.getName())
6 SpringContainer.addBd(bd);
```

特别说一下注意二，表示表示当前BeanDefinition在生成bean对象时，会通过调用MyFactoryBean的构造方法来生成，并传入UserMapper的Class对象。那么在生成MyFactoryBean时就会生成一个UserMapper接口对应的代理对象作为bean了。

到此为止，其实就完成了我们要解决的问题：**把Mybatis中的代理对象作为一个bean放入Spring容器中**。只是我们这里是用简单的JDK代理对象模拟的Mybatis中的代理对象，如果有时间，我们完全可以调用Mybatis中提供的方法区生成一个代理对象。这里就不花时间去介绍了。

Import

到这里，我们还有一个事情没有做，就是怎么真正的定义一个BeanDefinition，并把它**添加**到Spring中，上文说到我们要利用Import技术，比如可以这么实现：

定义如下类：

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {

 @Override
 public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefin
itionRegistry registry) {
 BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();
 AbstractBeanDefinition beanDefinition = builder.getBeanDefinition();
 beanDefinition.setBeanClass(MyFactoryBean.class);
 beanDefinition.getConstructorArgumentValues().addGenericArgumentValue(UserMapper.class);
 // 添加beanDefinition
  registry.registerBeanDefinition("my"+UserMapper.class.getSimpleName(), beanDefinition);
  }
}
```

并且在AppConfig上添加@Import注解：

```
@Import(MyImportBeanDefinitionRegistrar.class)
public class AppConfig {

```

这样在启动Spring时就会新增一个BeanDefinition，该BeanDefinition会生成一个MyFactoryBean对象，并且在生成MyFactoryBean对象时会传入UserMapper.class对象，通过MyFactoryBean内部的逻辑，相当于会自动生产一个UserMapper接口的代理对象作为一个bean。

总结

总结一下，通过我们的分析，我们要整合Spring和Mybatis，需要我们做的事情如下：

1. 定义一个MyFactoryBean
2. 定义一个MyImportBeanDefinitionRegistrar
3. 在AppConfig上添加一个注解@Import(MyImportBeanDefinitionRegistrar.class)

优化

这样就可以基本完成整合的需求了，当然还有两个点是可以优化的

第一，单独再定义一个@MyScan的注解，如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Import(MyImportBeanDefinitionRegistrar.class)
public @interface MyScan {
}
```

这样在AppConfig上直接使用@MyScan即可

第二，在MyImportBeanDefinitionRegistrar中，我们可以去扫描Mapper，在MyImportBeanDefinitionRegistrar我们可以通过AnnotationMetadata获取到对应的@MyScan注解，所以我们可以在@MyScan上设置一个value，用来指定待扫描的包路径。然后在MyImportBeanDefinitionRegistrar中获取所设置的包路径，然后扫描该路径下的所有Mapper，生成BeanDefinition，放入Spring容器中。

所以，到此为止，Spring整合Mybatis的核心原理就结束了，再次总结一下：

1. 定义一个MyFactoryBean，用来将Mybatis的代理对象生成一个bean对象
2. 定义一个MyImportBeanDefinitionRegistrar，用来生成不同Mapper对象的MyFactoryBean
3. 定义一个@MynScan，用来在启动Spring时执行MyImportBeanDefinitionRegistrar的逻辑，并指定包路径

以上这个三个要素分别对象org.mybatis.spring中的：

1. MapperFactoryBean
2. MapperScannerRegistrar
3. @MapperScan

文档：03-Spring整合Mybatis原理.note
链接：http://note.youdao.com/noteshare?id=c12df20198730f04c5d79936d5edbb30&sub=A0704FED0C6341B9BFBB17BE6CCBAE52